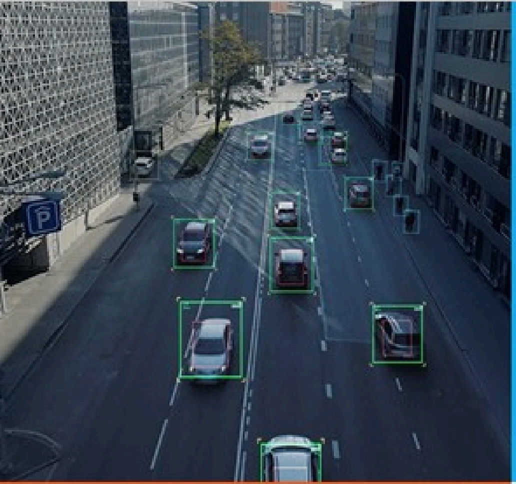


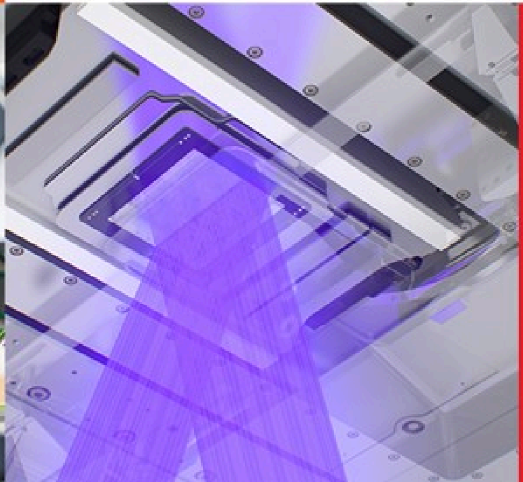
MECHATRONICS

MATHWARE

ASSEMBLY

SIOUX TECHNOLOGIES

ELECTRONICS



SOFTWARE

C++ Training

Value types

Pass by value

- Copies state of object
- Independent mutable state
- Simplifies thinking about MT

```
void do_something(std::vector<int>)
```

Move semantics facilitate performant value semantics

```cpp
std::vector<std::string> getPoems()
{
  std::vector<std::string> ans; // imagine ans contains data
  return ans;
}
int main(){
    std::vector<std::string> poems = getPoems();
}
```

return-value optimization

```
class GFG {
public:
    GFG() { std::cout << "Ctor"<<std::endl; }
    GFG(const GFG&){ std::cout << "Copy Ctor"<<std::endl; }
    GFG(GFG&&){ std::cout << "Move Ctor"<<std::endl; }
};
GFG func()
{
  return GFG(); // RVO example
}
int main()
{
    GFG G = func();
}
---
Ctor
```

Pass by reference

- Copy reference to object state
- Multiple references to same instance possible, race conditions
- Lifetime complexity

```
void do_something(int*, int size)
```

Value semantics

- Value semantics reduce complexity, language has core support for value semantics.
- Nr of bugs, ownership and concurrency are correlated.
- Core language legacy: Internal inheritance (e.g. vtable).

Library support

- std::variant<T1, T2>: closed type, open operation set polymorphism, single dispatch unlike typical double dispatch visitor implementations
- std::optional
- std::expected<V, E>
- std::any: void* with runtime safety
- …

std::variant

```
#include <iostream>
#include <variant>
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
using VariantType = std::variant<int, std::string>;
std::string get(VariantType v) {
    return std::visit(overloaded {
        [](int arg) { return std::to_string(arg); },
        [](std::string const& arg) { return arg;},
    }, v);
}
int main() {
    VariantType v = {"dro"};
    std::cout<<get(v)<<std::endl;
    v={10};
    std::cout<<get(v)<<std::endl;
}
```

Complexity

- Deep copying

  -- Produces a new, independent object; object (member) values are copied

  -- std::vector O(N) copies vs O(1) copies when using reference semantics

  -- Move semantics, copy elision reduces overhead.

- Copy on write

  -- Shared mutable state accessible from distinct contexts complicates the concurrency model. When should the shared object be destructed?

Internal inheritance

How does inheritance work?

```
struct C {
  virtual ~C(){}
  virtual do_something() {}
}
struct B : C {
  virtual ~B(){}
  virtual do_something() override {}
}
struct A : B {
  virtual ~A(){}
  do_something() override {}
}
```

vptr - mental model

```
Class A Object Layout:
+--------------------+
|      vptr          | --> Points to A's VTable
+--------------------+
| C's members        |
+--------------------+
| B's members        |
+--------------------+
| A's members        |
+--------------------+
Class B Object Layout:
+--------------------+
|      vptr          | --> Points to B's VTable
+--------------------+
| C's members        |
+--------------------+
| B's members        |
+--------------------+
Class C Object Layout:
+--------------------+
|      vptr          | --> Points to C's VTable
+--------------------+
| C's members        |
+--------------------+
```

vtable - mental model

VTable for Class A

```
+-----------------------------------+
| [0] Pointer to A::~A()            | --> A's Destructor
+-----------------------------------+
| [1] Pointer to A::do_something() | --> A's do_something()
+-----------------------------------+
```

Polymorphism

- Slicing occurs when vtable info is lost

```
Base& a = get_some_reference();
Base b = a;
```

- Create a new instance b of type B. vtable pointer is not copied.
- Slicing hinders the combination of value semantics and classic inheritance

exercises

exercises/value_semantics/ex1.cpp

Exercise that demos issues with internal inheritance and value semantics.

Value semantics for class hierarchy

Expressive, not loaded with c++ technicalities like references pointers etc.

```
Derived a;
Base &b = a;
auto c = b;
std::vector<Base> v; //value type Base
v.emplace_pack(a);
v.emplace_pack(b);
v.emplace_pack(c);
```

Goal

We want to generalize the concrete types to a common interface without any vtable like inheritance

Goal

```cpp
int main()
{
  std::vector<Animal> v;
  v.emplace_back(Dog());
  v.emplace_back(Cat());
  for (auto &a : v)
  {
    a.speak();
  }

}
```

Type erasure

```
struct Vtable
{
  void (*speak)(void *ptr);
  void (*destroy)(void *ptr);
};
```

Vtable stores function pointers that can operate on objects without knowing the concrete type at compile time

```cpp
template <class Concrete>
constexpr Vtable vtable_for
{
  [](void *ptr){ static_cast<Concrete*>(ptr)->speak(); },
  [](void *ptr){ delete static_cast<Concrete*>(ptr); },
};
```

- vtable_for is a variable template. C++14 feature.
- constexpr ensures compile-time instantiation.

```
struct Animal
{
  Animal(const T& t);
  ~Animal();
  void speak();
  void *m_concrete;
  Vtable const* m_vtable;
};
```

- Animal stores a void pointer to the concrete implementation
- Memory fragmentation. Pointer chasing.

```cpp
template<class T>
Animal(const T& t)
  : m_concrete(new T(t))
  , m_vtable(&vtable_for<T>)
{};
 ~Animal()
  {
    if (m_vtable)
      m_vtable->destroy(m_concrete);
  }

  void speak()
  {
    if (m_vtable)
      m_vtable->speak(m_concrete);
  }
```

- Templated constructor, instantiates vtable_for templated constexpr variable

```cpp
struct Dog
{
  void speak() { std::cout << "Woof\n"; }
};

struct Cat
{
  void speak() { std::cout << "Meow\n"; }
};
```

Other use-cases

- You can’t easily extend a class with interfaces
- Cleanup class hierarchy
- Template cleanup
- …

exercises

exercises/value_semantics/ex2.cpp

external inheritance

exercises/value_semantics/ex3.cpp

implement std::any

We bring **high-tech** to life